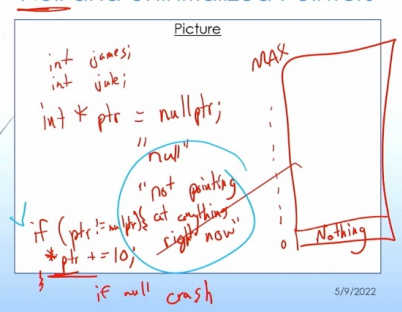
10 Null and Uninitialized Pointers

Picture



5/9/2022

11 Null and Uninitialized Pointers

- A **null pointer** has value `0x0` (i.e. it points to address 0)
  - No objects are allowed to live at address 0.
  - A null pointer is interpreted as "not pointing to anything".
  - Dereference a null pointer → **runtime error** (usually).
  - Declare a null pointer like this: `int *ptr = nullptr;`
- Just like any other variable of primitive type, an uninitialized pointer has no value in particular.
  - It's pointing at some random place in memory!
  - Dereference an uninitialized pointer → **undefined behavior**.
    - Maybe it crashes? If the pointer is pointing to memory your program isn't allowed to use, you might get a **segmentation fault**.
    - Maybe you read some random memory and get junk values?
    - Maybe you write some random memory and mess up other stuff.

Slides

18 Pass-by-Pointer (by Value)

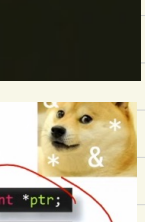
address (copy)

```
void addOne(int *x) {
  *x += 1; // adds 1 to whatever x points to, which
           // is z in this example, even though the
           // name z is not in scope here.
           // We used the address of z to get to it.
}

int main() {
  int z = 1;
  int *ptr = &z; // ptr "points to" z

  *ptr += 1; // adds 1 to z, without using the name z

  addOne(&z); // adds one to z
  addOne(ptr); // same thing
}
```



X Slides

21 Exercise: Pointer Trap

- L03.4_pointers on Lobster.


```
int * getAddress(int x) {
  return &x; // It's a trap!
}
void printAnInt(int anInt) {
  cout << anInt << endl;
}
int main() {
  int a = 3;
  int *ptr = getAddress(a);

  printAnInt(42);

  // should print 3, right???
  cout << *ptr << endl;
}
```

Clicker Question

Why is it a trap?

A) Can't return pointers from functions

B) `anInt` became a reference to `x`

C) The lifetime of the parameter `x` ended before `*ptr` was used

D) `ptr` became uninitialized when `printAnInt` was called

5/9/2022

```
1  #include <iostream>
2  using namespace std;
3
4  int * getAddress(int &x) {
5    return &x;  // It's a trap!
6  }
7
8  void printAnInt(int someInt) {
9    cout << someInt << endl;
10 }
11
12 int main() {
13   int a = 3;
14   int *ptr = getAddress(a);
15   printAnInt(42);
16   cout << *ptr << endl;
17 }
```

use this way to solve the problem. If we pass by value, the address we Got from this function would be the Address of the memory in the stack And will dead after the function has Been called. However, when we use Pass by reference, when call the function, We will not actually creat a new memory Address to x, instead the x will be another name of the variable a.then the return of the function will be definitely the address of the variable "a" and will still alive after the function had been called.

23 So Many * and &

- Used to specify a type…
  - * means it's a pointer `int *ptr;`
  - & means it's a reference `int &ref;`
- Used as an operator in an expression…
  - * means get object at an address `cout << *ptr << endl;`
  - & means take address of an object `cout << &x << endl;`

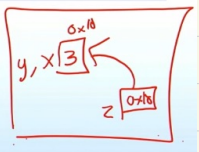
deref

5/9/2022

24 References vs. Pointers

References	Pointers
An alias for an object	Stores address of an object

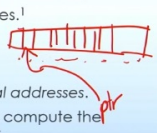
```
int main() {
  int x = 3;
  int &y = x;
  int *z = &x;
}
```



- You can change where a pointer points.
- You cannot re-bind a reference!

What can you do with pointers?

- Work with objects *indirectly*.
  - "Simulate" reference semantics.
  - Use objects across different scopes.
  - Enable subtype polymorphism.[1]
  - Keep track of objects in dynamic memory.[1]
  - Implement linked data structures.[1]
- **Work with *arrays* of objects.**
  - Objects in arrays have *sequential addresses*.
  - We can do *pointer arithmetic* to compute the address of the element we want.



1 We'll look at these later in the course.

5/9/2022